# CS 300 Pseudocode Document

Chuke Rupert

CS300-T4190

9 April, 2023

## Function Signatures

// course object Struct declaration

Struct Course{

string number

string title

vector<string> preReqs

}

// inspect line for errors [O(2)]

Function checkLine(lineStr\_){

If lineStr\_ == "(course number), (title), ([0..] preReqs)"

return True

return False

}

// Turn readLine string into Course object [O(7+2\*prereqs)]

Function makeCourse(string currLine)

string currLine, cNum, cTitle

// check for errors

If !checkLine(currLine){

THROW error

}

// parse string elements into variables

cNum = currLine.Substring(to first ',')

cTitle = currLine.Substring(first ', ' to second ',' or '\n')

While (currLine != '\n'){

cPreReq.Add(currLine.Substring(up to next ',' or '\n'))

}

// return new Course

Return new Course(cNum, cTitle, cPreReq))

}

// Open file and generate vector of Course objects [O(10n+3)]

Function getCourses(string fileName){

Stream f = open fileName in read

vector<Course> courseVec;

While (f != endOfFile){

// currLine becomes next line in file

currLine = f.readLine

// generate Course object and add to vector

courseVec.Add(makeCourse(currLine))

}

Return courseVec

}

// print a course information

Function displayCourseInfo(thisCourse){

print ("(number) | (Title) | ([preReq1, preReq2,...]) (Endline)")

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// vector data structure methods

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// print a course information

Function vecPrintCourse(courseNum, courseVec){

For each Course in courseVec{

if courseNum == Course.number{

displayCourseInfo(Course)

Return

}

}

// feedback if course not found

print ("Course not found\n")

}

// print all courses in order

Function vecCourseList(courseVec){

//sort vector by department then with radix sort

sortedVec = ModifiedRadixSort(courseVec, courseVec.size())

For each Course in sortedVec{

displayCourseInfo(Course);

}

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Hashtable methods

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Hashing function for insertion/search [O(2)]

Function Hash(courseNum){

convert last 3 digits in courseNum to integer

Return (integer courseNum) % (num of buckets)

}

// Insert course into Hashtable [O(2i+10)]

Function hashInsert(hashTable, course){

IF (hashSearch(hashTable, course.number) == Null){

GET bucketList from hastTable(Hash(course.number))

new Node(course)

append Node to bucketList

}

}

// Search Hashtable for course number [O(2i+4)]

Function hashSearch(hashTable, courseNum){

GET bucketList from hastTable(Hash(course.number))

For each Node in bucketList{

IF course.number == courseNum

Return Course

}

Return Null

}

// Print a course from hastTable

Function coursePrintHash(hashTable, courseNum){

displayCourseInfo(hashSearch(hashTable, courseNum))

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Binary Search Tree methods

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// BST Node structure

Struct Node{

Course course

Node\* left

Node\* right

}

// BST declaration

Class CourseTree{

Node\* root\_ = Null

}

// add nodes to Tree [O(4log(n)+4)]

Function addNode(Tree, courseToAdd){

// Empty location found

IF Tree.root\_ == Null{

Tree.root\_ = new Node(courseToAdd)

}

ELSE{

SET currentNode equal to Tree.root\_

WHILE currentNode is not null:

IF node is less than currentNode:

SET currentNode equal to left child of currentNode

ELSE IF node is greater than currentNode:

SET currentNode equal to right child of currentNode

ELSE:

RETURN

SET currentNode equal to node

}

}

// Search Tree for course [O(5log(n)+2)]

Function treeSearch(Tree, courseNum){

currentNode\* = Tree.root\_

WHILE currentNode != NULL{

IF currentNode.Course.number == courseNum

RETURN currentNode.Course

ELSEIF courseNum < currentNode.Course.number

currentNode = left child

ELSE{

currentNode = right child

}

}

RETURN Empty Course

}

// Print one course information [O(5log(n)+5)]

Function coursePrintTree(Tree, courseNum){

new Course = treeSearch(Tree, courseNum)

IF found

displayCourseInfo(Course)

ELSE

print ("Course not found\n")

}

// Traverse tree in order and print all courses

Function inOrderDisplay(node){

IF (node->left != Null)

inOrderDisplay(node->left)

displayCourseInfo(node->course)

IF (node->right != Null)

inOrderDisplay(node->right)

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Main method

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Main{

int Choice

print salutation/program info

While True{

print menu to user:

print ("Please select an option from below:\n")

print ("1: Load data\n")

print ("2: Print Course List")

print ("3: Print Course information")

print ("4: Exit Program")

print ("Enter your choice: ")

choice = userinput

IF choice is not int 1-4

print ("Invalid input! please select a value between 1-4")

Continue

Switch Choice:

// case 1: load data

case 1:

// Load data from file into vector

courses\_Vector = getCourses("CourseData.csv")

// initialize Tree and table

courses\_Tree = new CourseTree

courses\_Table = new hashTable(25)

// load data into tree and table

For each Course in courses\_Vector{

addNode(courses\_Tree, Course)

hashInsert(courses\_Table, Course)

}

print ("data load complete!\n\n")

// case 2: print all courses

case 2:

IF (!courses\_Tree){

print ("Data not loaded")

Continue}

print ("Course List: \n")

inOrderDisplay(courses\_Tree.root\_)

print ("End of List\n\n")

// case 3: search a course and print information

case 3:

IF (!courses\_Tree){

print ("Data not loaded")

Continue}

print ("Please type the course number of the course you want to see:")

string toSearch = userinput

coursePrintHash(courses\_Table, toSearch)

print ("\n\n")

// case 4: Exit Program

case 4:

print ("Goodbye!\n")

RETURN 0

}

}

## Runtime Analysis

Open File in vector:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| 34 Stream f = open fileName in read | 1 | 1 | 1 |
| **35** vector<Course> courseVec; | 1 | 1 | 1 |
| 36 While (f != endOfFile){ | 1 | n+1 | n+1 |
| 38 currLine = f.readLine | 1 | n | n |
| 40 courseVec.Add(makeCourse(currLine)) | 7+ | n | (7+2x)n |
| 42 Return courseVec | 1 | 1 | 1 |
| **Total Cost** | | | (~9)n+4 |
| **Runtime** | | | O(n) |

Open File in hash: (theoretical)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Stream f = open fileName in read | 1 | 1 | 1 |
| C\_Table = new hashTable | 1 | 1 | 1 |
| While (f != endOfFile){ | 1 | n+1 | n+1 |
| currLine = f.readLine | 1 | n | n |
| hashInsert(C\_Table, makeCourse(currLine)) | 7+12 | n | 19n |
| **Total Cost** | | | 21n+3 |
| **Runtime** | | | O(n) |

Open File in hash: (as-written)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| 200 courses\_Vector = getCourses("CourseData.csv") | 9n+4 | 1 | 9n+4 |
| 203 courses\_Table = new hashTable(25) | 1 | 1 | 1 |
| 205 For each Course in courses\_Vector{ | 1 | n | n |
| 209 hashInsert(courses\_Table, Course) | 12 | n | 12n |
| **Total Cost** | | | 22n+5 |
| **Runtime** | | | O(n) |

Open File in BST: (as-written)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| 200 courses\_Vector = getCourses("CourseData.csv") | 9n+4 | 1 | 9n+4 |
| 202 courses\_Tree = new CourseTree | 1 | 1 | 1 |
| 205 For each Course in courses\_Vector{ | 1 | n | n |
| 206 addNode(courses\_Tree, Course) | 4log(n)+4 | n | 4nlog(n)+4 |
| **Total Cost** | | | (4log(n)+10)n + 9 |
| **Runtime** | | | O(nlog(n)) |

Open File in BST: (theoretical)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Stream f = open fileName in read | 1 | 1 | 1 |
| C\_Tree = new CourseTree | 1 | 1 | 1 |
| While (f != endOfFile){ | 1 | n+1 | n+1 |
| currLine = f.readLine | 1 | n | n |
| addNode(C\_Tree, makeCourse(currLine)) | 7+4log(n)+4 | n | (11+4log(n))n |
| **Total Cost** | | | (13+4log(n))n + 3 |
| **Runtime** | | | O(nlog(n)) |

## Structure comparison

With regard to generating each structure, the apparently weakest choice is the Binary Search Tree, assuming a somewhat large n (>7 in this case), with a runtime of O(n\*log(n)). Meanwhile, both the vector and the hashTable have a runtime of O(n), though the vector will populate marginally faster as it doesn’t require hashing or searching functions prior to appending new courses. All three structures use the same methods for opening the target file and checking the file for errors, each with runtimes of O(1) and O(n) respectively. As a result of all this, for the sole use of storing the course information data, a vector would be my recommendation.

With that said, the advantages of the hashTable and Tree are revealed when searching and sorting the courses for various purposes. For example, searching the hashTable for a specific course has a worst case runtime of O(1), far superior to the O(n) runtime for the vector data structure, and somewhat better than the O(log(n)) for the Tree. Also worth considering is the ability of each structure to print the entire course list in sorted order. The hashTable here is clearly weakest as it cannot sort it’s data without first returning it to another structure. The Tree structure is easiest to do this with as it is sorted every time a node is added, giving it a strict O(n) runtime. Finally, the vector has to be sorted first before it is printed, and can have anywhere from a O(n) to O(n2) runtime depending on the sorting algorithm used. Given these additional considerations/requirements from ABCU, I would recommend the use of a Binary Search Tree. While it does have the greatest demand for time/resources when initializing, that process should only realistically occur once each time the program is run. Meanwhile, it has the fastest and simplest method for printing the list of all courses, and has a reasonably fast search time for a specific course.